Fast Universal Background Model (UBM) Training on GPUs using Compute Unified Device Architecture (CUDA)

M. Azhari, C. Ergün.
Computer Engineering Department, Eastern Mediterranean University

Abstract—Universal Background Modeling (UBM) is an alternative hypothesized modeling that is used extensively in Speaker Verification (SV) systems. Training the background models from large speech data requires a significant amount of memory and computational load. In this paper a parallel implementation of speaker verification system based on Gaussian Mixture Modeling – Universal Background Modeling (GMM – UBM) designed for many-core architecture of NVIDIA’s Graphics Processing Units (GPU) using CUDA single instruction multiple threads (SIMT) model is presented. CUDA implementation of these algorithms is designed in such a way that the speed of computation of the algorithm increases with number of GPU cores. In this experiment 30 times speedup for k-means clustering and 16 times speedup for Expectation Maximization (EM) was achieved for an input of about 350K frames of 16 dimensions and 1024 mixtures on GeForce GTX 570 (NVIDIA Fermi Series) with 480 cores when compared to a single threaded implementation on the traditional CPU.

Index Term—Compute Unified Device Architecture (CUDA), Expectation Maximization (EM), Speaker Verification, Universal Background Model (UBM)

I. INTRODUCTION

UNIVERSAL BACKGROUND MODELING is an approach to approximate the imposter model in speaker verification systems. In this approach, pooled training data from a large number of speakers is utilized to construct a large mixture model. The likelihood of background speakers used to form reference. The UBM model parameters are also trained by estimating parameters $\lambda$ of the GMM that matches the distribution of the training feature vectors using training speech from a speaker. The most popular, well-established and robust method is the Maximum Likelihood (ML) which iteratively refines the GMM parameters. One important property of ML estimation is that for a large enough set of training, the model estimate converges to the true model parameters as the data length grows. Since there is not closed-form for computing the ML estimate, an iterative technique known as Expectation Maximization is used [1].

Optimization of the SV systems can help to decrease development time. It should be kept in mind that the mixture based modeling techniques can be easily optimized using parallel implementation. The existence of a massively parallel computing technology such as CUDA has changed the face of computing in the past years. Currently having some of the most efficient ratings on Green500 list [2] indicates that today’s GPUs are more suited for scientific computing with less power consumption. In this paper a research on parallelizing UBM parameter estimation of SV is performed. In general, a main SV system includes front end processing (noise cancellation, feature extraction and selection), background model generation, adaptation and testing phases. In this paper the focus will be on background model parameters generation for Mel-Frequency Cepstrum Coefficients (MFCC) data and parallelization of units such as MFCC, k-means and Expectation Maximization algorithm. The contribution is on two different architectures of NVIDIA GPUs namely GTX 285 (GT200 Architecture) and GTX 570 (Fermi Architecture). The speedup results of the latter with the traditional CPU architecture, both in single threaded and multi-core implementations will be compared. It will be observed how the use of even naïve GPU implementation can result in faster calculations and how different hardware architectures can affect the speed. Also it will be explained how the techniques in GPU software implementation directly target optimality of the performance. GPUs have a very high floating point calculation throughput (+1 TFlop/s peak performance as opposed to a Core i7 CPU - theoretical peak performance of 109 GFlops) [3]. In this paper, the nature of the problem was introduced with double precision floating points. The mentioned GPUs perform double precision operations theoretically 175 GFlops and 86 GFlops for GTX285 and GTX570 respectively. Considering the size of the problem (about 350K samples of 16 dimensions) the solution is not memory-bound and it comes down to the computation limit. For the time being, shared memory approaches were not stepped into as it requires a more programatically complex implementation and it suffers from issues such as memory coalescing [4]. The rest of the paper is organized in the following manner. In section II, the MFCC, k-means and EM algorithms are described to generate UBM parameters. In section III, the architecture of CUDA Fermi

M. A. is with Computer Engineering Department, Eastern Mediterranean University, Famagusta, Mersin 10 Turkey (mohammad.azhari@emu.edu.tr).
C. E. is with Computer Engineering Department, Eastern Mediterranean University, Famagusta, Mersin 10 Turkey (cem.ergun@emu.edu.tr).
will be presented. In section IV the design of latter algorithms is discussed both for CPU Parallel and CUDA. The experimentation and results of different architecture models and analysis of performance boosts are presented in section V. Section VI concludes the paper and presents some future work.

II. FRONT-END MODULE AND UBM GENERATION

A. Mel-Frequency Cepstrum Coefficients

Mel-frequency Cepstrum coefficients are well known features used to describe speech signal. They are based on the evidence that the information carried by low-frequency components of the speech signal is phonetically more important for humans than the information carried by high frequency components [5]. The Mel scale reflects this by using a nonlinear warping of the frequency scale, i.e., by reducing the frequency resolution as the frequency increases. The process of extracting MFCC from continuous speech is illustrated in Fig 1.

![Fig. 1. Block diagram of the MFCC computation process [15]](image)

In order to calculate the MFCC of a speech segment, the first block of the figure obtains the Discrete Fourier Transform (DFT) of the segment. At this step, the length of this sequence is increased from \( N \) to \( 2N \) by inserting zeros to improve the resolution in the frequency domain (zero padding). Then the segment is transformed into the frequency domain as

\[
S[k] = \sum_{n=0}^{2N-1} s[n]e^{-\frac{2\pi kn}{2N}} \quad k = 1, \ldots, N \quad (1)
\]

where \( s[n] \) is the windowed speech segment of length \( N \). The power spectrum is computed by taking the absolute value of the DFT. This is represented as the second block in Fig 1. Then the power spectrum is passed through each Mel filter of the filter bank for calculating the output power of each filter.

The final processing step is to apply Discrete Cosine Transform (DCT) to the log of the filter-bank coefficients, yielding the MFCC parameters as

\[
C_m = \frac{1}{N_f} \sum_{j=1}^{N_f} \log(e_j) \cos\left(\frac{mN_f}{N_f} \left(\frac{j-1}{2}\right)\right) \quad m = 1, \ldots, D \quad (2)
\]

where \( D \) is the number of MFCC parameters calculated for each frame. In (2), it can be seen that \( C_0 \) represents the average log energy of the spectrum. Since it is preferred to use a feature set which is not susceptible to varying background noise and loudness of speech, \( C_0 \) is omitted, resulting in a \( D \)-dimensional MFCC feature vector as,

\[
\vec{x} = [C_1, C_2, \ldots, C_D] \quad (3)
\]

B. k-means Clustering

k-means clustering is a method that helps to accelerate convergence [6]. In this method with a set of observations \( \{x_1, x_2, \ldots, x_n\} \), each of which a vector of \( d \) dimensions are clustered into \( k \) sets \( (k \leq n) \) \( S = \{S_1, S_2, \ldots, S_k\} \) such that the sum of square in each cluster is minimized:

\[
\arg \min_{\mu_i} \sum_{j \in S_i} \sum_{k=1}^{d} (x_{ijk} - \mu_i) \quad (4)
\]

where \( \mu_i \) is the mean of point in \( S_i \).

C. Expectation Maximization Algorithm

Expectation-maximization (EM) algorithm is a method that iteratively estimates the likelihood [7]. It is similar to the k-means clustering algorithm for Gaussian mixtures since they both look for the center of clusters and refinement is done iteratively.

For a given set of \( T \) observation vectors \( X = \{x_1, x_2, \ldots, x_T\} \) and assuming \( x_i \) vectors are independent and identically distributed (iid), the best fitting model \( \lambda \) is the one that maximizes,

\[
p(X | \lambda) = \prod_{t=1}^{T} p(x_t | \lambda). \quad (5)
\]

This is a non-linear problem; therefore \( \lambda \) cannot be directly calculated. The EM algorithm consists of the following steps:
1. Choose an initial model \( \lambda \).
2. Find a new model \( \lambda' \) so that \( p(X|\lambda') > p(X|\lambda) \).
3. Repeat step 2 until the difference
4. \( p(X|\lambda') \cdot p(X|\lambda) \) has reached a convergence threshold or you have reached the maximum number of iterations.

Alg. 1. Expectation Maximization Procedure

In each EM iteration, the ML estimates for the means, variances and weights (a priori mixture probability) for a particular speaker model are computed as follows:

Mixture Weight:
\[
\overline{m}_i = \frac{1}{T} \sum_{t=1}^{T} P(i|x_t, \lambda)
\]  \hspace{1cm} (6)

Means:
\[
\overline{\mu}_i = \frac{\sum_{t=1}^{T} P(i|x_t, \lambda) x_t}{\sum_{t=1}^{T} P(i|x_t, \lambda)}
\]  \hspace{1cm} (7)

Variances:
\[
\overline{\sigma}_i^2 = \frac{\sum_{t=1}^{T} P(i|x_t, \lambda) x_t^2}{\sum_{t=1}^{T} P(i|x_t, \lambda)} - \overline{\mu}_i^2
\]  \hspace{1cm} (8)

where \( \overline{m}_i \), \( \overline{\mu}_i \) and \( \overline{\sigma}_i^2 \) are updated values of \( m_i \), \( \mu_i \) and \( \sigma_i^2 \) respectively, and \( \overline{\sigma}_i^2 \), \( \overline{\mu}_i \) and \( x_t \) refer to arbitrary elements of the vectors in \( \overline{\sigma}_i^2 \), \( \overline{\mu}_i \) and \( x_t \) respectively and, \( x_t \) is the shorthand for dialog \( x_1, x_2 \). The posterior probability for the \( i \)th acoustic class is given by,
\[
P(i|x_t, \lambda) = \frac{m_i b_i(x_t)}{\sum_{i=1}^{M} m_i b_i(x_t)}
\]  \hspace{1cm} (9)

One of the parameters that should be selected before training a Gaussian mixture speaker model is the order \( M \) of the mixture density. The model parameters must also be initialized prior to the EM algorithm. These selections are experimentally determined for a given task. In this experiment, the order of mixture density \( M \) is 1024. The EM algorithm is guaranteed to find a locally optimal maximum likelihood model regardless of the starting point. However it may have several local maxima and different starting models can lead to different local maxima [8]. Alternatively, k-means algorithm can be used to perform a clustering of the training data into \( M \) acoustic classes [9]. The \( M \) initial estimates for the mixture means and the covariance matrices can be selected as the mean and the covariance of each cluster.

III. COMPUTE UNIFIED DEVICE ARCHITECTURE

CUDA [10] is the parallel computing architecture developed by NVIDIA and is the computing engine in NVIDIA graphical processing units (GPUs). This architecture is available through different programming languages and supports other application programming interfaces, such as CUDA FORTRAN, OpenCL, and DirectCompute. CUDA helps to solve many complex computational problems in a more efficient way than on a CPU. The CUDA parallel programming model is designed to overcome the challenge of developing application software that transparently scales its parallelism while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

CUDA has several advantages over traditional general purpose computation on GPUs [11]. One of them is that the GPU code can access different addresses in memory. Another advantage is availability of shared memory which is a locally accessible memory that is shared between a group of threads. This helps to reduce the global memory accesses and as a result providing a higher bandwidth.

However, there are some limitations. One is that the data transfer between the device and the host memory is slower that within-device memory transfers. Threads are activated in groups of 32, with thousands of them running in total. Although [12] discusses that this may not always be the case, including [13] and current implementation in which a small number of threads per block group were used to get better results. Another limitation is that CUDA technology is only available for NVIDIA GPUs. It only supports round-to-nearest mode of IEEE 754 [14] standard for double precision calculations.

On CUDA architecture, the problems are divided into sub-problems and each sub-problem into finer pieces that can cooperatively run in parallel by all threads within the block. Each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Fig. 2:

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as
illustrated by Fig. 3. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

![Grid of Thread Blocks](image)

Fig. 3. Grid of Thread Blocks [10]

Each running block is divided into Single Instruction Multiple Threads groups called warps. The size of these warps are equal. The running warps are scheduled in a timely manner (time-sliced). The thread scheduler switches between warps to maximize the use of the multiprocessors computational resources.

IV. IMPLEMENTATION AND DESIGN

A. MFCC

Due to the nature of speech data, it is very normal to have unvoiced sound segments. Here, Voice Activation Data (VAD) is used to discard silence periods [15]. The other front-end operations like segmentation and windowing are also applied to each frame in order to produce the corresponding feature set. The complexity of MFCC extraction is mainly dominated by Discrete Fourier Transform (DFT) of each window segment which is $O(N^2)$ where $N$ is the window size. In this study only the DFT step of MFCC is parallelized as shown in Fig. 1. The CPU DFT pseudo-code for window size of 240 is as illustrated in Alg. 2.

In order to have load balancing, the execution of the loop at line 5 of Alg. 2 was split into I number of cores. Each core is responsible for executing operations of a range of windows size frames. In current case an Intel® Core™ i7-920 [3] CPU was used to perform the parallel task on 4 cores. Note that the cosine and sine values are pre-computed to decrease the redundancy of the calculations.

1. $\text{Cores} = 4$;
2. $N = 240$; // Winosize
3. $\text{RangeSize} = N / \text{Cores}$;
4. For core $I$ (concurrently)
5. For $k$ from $[\text{RangeSize} * I]$ to $[\text{RangeSize} * (I + 1)]$
6. Initialize Real and Imaginary variables
7. For $n$ from 0 to $2 * N$
8. Calculate Real & Imaginary
9. Calculate DFT

Alg. 2. CPU DFT Parallel

In this algorithm, the time complexity is $O(N^2)$. On CPU Parallelism, the complexity is only divided by a number much smaller than $N$. Therefore the complexity remains the same. The CUDA DFT implementation in nature also follows the CPU model with the exception that each iteration of loop $k$ is run on a separate thread. In this case the complexity is reduced to $O(N)$. There are also memory transfers between host and CUDA device as shown in Alg. 3.

![CUDA DFT Parallel](image)

Alg. 3. CUDA DFT Parallel

B. k-means Clustering

The k-means clustering algorithm pseudo-code in a nutshell is as shown in Alg. 4. Since initial selection of mean is random, mean vectors of size $M$ mixtures by $D$ dimensions were chosen from the feature vectors.

1. Random initialization of mean vectors.
2. $T=\text{number of frames}$;
3. $M=\text{order of mixtures}$
4. $D=\text{number of dimensions}$;
5. For each frame $T$
6. For each mixture $M$
7. For each feature $D$
8. Calculate minimum distance
9. Calculate average of selected feature vectors belonging to the same centroid.

Alg. 4. k-means Pseudo-Code

In parallel implementation it is advised to give more work load to each processor to maximize the utilization. In the case of k-means clustering, the major loop that is parallelizable is loop $T$ which includes inner loop $M$ and $D$. The computation complexity of k-means clustering is $O(p^{\log n})$. Since $k$ and $d$ are fixed, the problem will come down to $n$ number of entities to be clustered [15]. In both CPU and GPU approach
the most outer loop was parallelized which in this case is \( T \),
given the fact that the distances can be calculated concurrently. For CUDA device implementation the calculations were performed for each feature vector simultaneously. As a result the complexity will be \( O(n^d \log n') \) where \( n' < n \). Additionally, there are memory transfers from and to device. The order of CUDA actions for k-means algorithm is shown below:

1. Random initialization of mean vectors.
2. \( T = \) number of frames;
3. \( M = \) order of mixtures
4. \( D = \) number of dimensions;
5. Transfer initialized random mean vectors to device.
6. Transfer feature vectors to device.
7. Allocate label memory on device.
8. Allocate mean average memory on device.
9. Allocate distance memory on device.
10. Perform k-means on CUDA device with grid size of \( T/20+1 \) and block size of 20.
11. Transfer label memory back to host.
12. Transfer mean-average memory back to host.
13. Calculate average of selected feature vectors belonging to the same centroid.

Alg. 5. k-means CUDA

Note that all of 2-dimensional jagged arrays are first converted into 1-dimensional arrays for device memory allocation.

C. Expectation Maximization

As explained in section II. C., the first step of expectation maximization is to initialize \( \lambda \) parameters mixture weights (6), means (7) and variances (8). The computation complexity for background speaker model is directly extracted from Alg. 6 as \( O(N_p MD) \) where \( N_p \) is the number of feature vectors each having dimension of \( D \) and each speaker having \( M \) mixtures. In order to prevent redundant calculations, first the common part of those parameters is calculated which is:

\[
\sum_{i=1}^{T} P(i \mid \bar{x}_i, \lambda)
\]

(10)

where \( i \) is the number of mixtures. Following the (9), the denominator of (9) is calculated, so later it can be applied for the posterior probability. Then it is continued with calculating posterior probability, sum of means and sum of variances. Then (6), (7) and (8) were applied to calculate the updated values \( \bar{m}_i \), \( \bar{\mu_i} \) and \( \bar{\sigma_i^2} \). This concludes one iteration of EM Algorithm. In this implementation the iterations are repeated 50 times. The pseudo-code for EM algorithm is illustrated in Alg. 6.

Alg. 6. Expectation Maximization Pseudo-Code for CPU

The mentionable points of parallelization in Alg. 6 is where loops of size \( T \) exists. Since the convergence iterations are dependent and have to be executed sequentially, the most dominant parallelizable loop is \( T \). In case of CPU parallelism, a similar method is applied as seen in Alg. 2 to distribute the load. In a nutshell the following tasks are performed in CPU Parallel mode:

Alg. 7. EM Pseudo-Code for Parallel CPU

The \( D \) loops are left sequential because in the small loops the cost of creating multiple threads is more than the calculation cost in a single thread. On the other hand the \( K \) loop is not parallelized for the fact that the inner loop \( T \) is big.
enough to cover the cost of thread activation.

In CUDA Parallel mode the procedures were sliced into smaller groups to handle the number of threads better. Since the CUDA code is SIMT all threads run identical code only on different parts of the memory. For example in the case of initializing sum of means and sum of variances it is only needed to have \( D \) threads while for updating them \( T \) threads are activated. This calls for task separation. It can be understood from Alg. 7 that for EM algorithm there are more memory transfers/allocations are involved. It is also known that the CUDA code will run on device as kernels. Kernels are C extension functions of CUDA technology. They can be launched from the host code to run on many CUDA threads. The importance of grid size and block size will be discussed in section V. The following is the EM algorithm designed to run on CUDA device:

```
1. Initialize mean, variance and weight values.
2. \( T \) = Number of feature vectors. (around 350K)
3. \( M \) = Order of mixtures (1024)
4. \( D \) = Number of dimensions in a feature vector (16)
5. Allocate memory on device and transfer the initialized mean, variance and weight.
6. Transfer feature vectors to device memory.
7. For iteration between 0 and 50 {
8. Calculate Determinant of Sigma on 2 blocks of size \( M/2 \);
9. Calculate Sum p denominator on \( T / 20 + 1 \) blocks of size 20;
10. For \( k \) between 0 and \( M \)
11. Initialize SumMean and SumVariance on 1 block of size \( D \);
12. Calculate SumMean and SumVariance on \( T / 32 + 1 \) blocks of size 32;
13. Update Mean & Variance on 1 block of size \( D \);
14. Update Weight on 1 block of size 1;
15. Transfer mean, weight and variance from device to host memory
```

Alg. 8. EM Pseudo-Code for Parallel CUDA

The complexity of this EM algorithm is reduced to \( O(MD) \).

V. EXPERIMENTS

The parallel algorithms were implemented using CUDA version 3.2. The experiments were performed on a PC with GTX 285 and GTX 570 GPUs and an Intel® Core™ i7-920 CPU. The CPU has 4 cores (8 hyper-threads) running at 2.66 GHz. The main memory is 3 GB (DDR3-1600) with the peak bandwidth of 12.8 GB/sec. The specification of used GPUs can be seen in Table I.

<table>
<thead>
<tr>
<th>GPU</th>
<th>Cores</th>
<th>DRAM</th>
<th>Processor Clock</th>
<th>Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 285</td>
<td>240</td>
<td>1 GB</td>
<td>1.47 GHz</td>
<td>159 GB/s</td>
</tr>
<tr>
<td>GTX 570</td>
<td>480</td>
<td>1.25 GB</td>
<td>1.46 GHz</td>
<td>152 GB/s</td>
</tr>
</tbody>
</table>

The results of calculations were presented while omitting the file read/write operations to concentrate on algorithm’s speedup. The results were performed on around data size of 350,000 feature vectors belonging to 92 female speakers. The Table II shows the actual times taken to perform MFCC and k-means algorithm in seconds.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CPU</th>
<th>CPU Parallel</th>
<th>CUDA (GTX 285)</th>
<th>CUDA (GTX 570)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFCC</td>
<td>216.6</td>
<td>111.47</td>
<td>310</td>
<td>254</td>
</tr>
<tr>
<td>k-means</td>
<td>59.36</td>
<td>14.25</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

It can be observed that the experimented MFCC algorithm has poor performance compared to traditional CPU algorithm. The reasons for this are the following:

- The only parallelized function within MFCC was discrete fourier transform algorithm. Although the complexity is reduced in theory, given the size of the problem for a single DFT run, it is not feasible to distribute the subproblem into too many threads. For this experiment the size of outer loop was 240 and hence 240 threads were activated, but because of the low occupancy level the time needed to read/write into device global memory affected the performance.
- The other reason is low throughput of memory transfer between host and device. The DFT algorithm is called almost 350,000 times and each time it copies the window frames from host to device and copies the results of DFT back to host memory.

In k-means algorithm a considerable performance improvement can be seen when data-parallelism is compared to single threaded CPU. However the time taken on both GPU architectures is the same. One reason for this can be lack of use of shared memory to benefit from the architecture specific advantages. This may also indicate that current algorithm suffers from memory partition camping [16]. Global memory accesses go through partitions. Successive 256-byte regions of global memory are assigned to successive partitions. The problem of partition camping is when global memory accesses at an instant use a subset of partitions. This may hide the true potential of speedup scalability across cores. For optimal performance GPU accesses should be distributed evenly among partitions.

The results of EM algorithm are measured in parts to show the execution times more clearly. Table III shows the EM time in seconds.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CPU</th>
<th>CPU Parallel</th>
<th>GTX 285</th>
<th>GTX 570</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM_P Per EM Iteration</td>
<td>56</td>
<td>17</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Updating Mean,Var &amp; Weight Per EM Iteration</td>
<td>159</td>
<td>45.27</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>EM total per iteration</td>
<td>215</td>
<td>62.27</td>
<td>30</td>
<td>12</td>
</tr>
</tbody>
</table>

Again, faster calculations on more cores and further on many cores of GPUs can be seen. Table IV shows the speedup of all algorithms on different systems. The speedups are...
calculated using \( t1 / t2 \) where \( t1 \) is the single threaded CPU time and \( t2 \) is the target parallel model time. Additionally, Fig. 4 shows the graph corresponding to Table IV.

<table>
<thead>
<tr>
<th>Stage</th>
<th>CPU</th>
<th>CPU Parallel</th>
<th>GTX 285</th>
<th>GTX 570</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFCC</td>
<td>1</td>
<td>1.94</td>
<td>0.69</td>
<td>0.85</td>
</tr>
<tr>
<td>k-means</td>
<td>1</td>
<td>4.16</td>
<td>29.68</td>
<td>29.68</td>
</tr>
<tr>
<td>EM</td>
<td>1</td>
<td>3.45</td>
<td>7.16</td>
<td>17.91</td>
</tr>
</tbody>
</table>

It is observed that the increase in the number of cores will decrease the calculation time. However, the technique of choosing the correct grid and block sizes should be taken into consideration. Too little threads in a block may race against the occupancy factor of the active blocks [10]. Too many threads in a block may prevent the use of shared memory since there is a limit of 16KB on GTX 285 and 48KB on GTX 570 models per block. In this study, a naïve GPU code that uses the global memory was provided. The number of threads used per CUDA block varies from 1 to 32. That is because current implementations performed better in low block dimension sizes. According to [12] in some cases such as [13] a better performance may be achieved in very low block sizes.

VI. CONCLUSION

In this paper, a CUDA based implementation of DFT, k-means and EM algorithms were introduced. Also, the CPU parallelism of the mentioned algorithms for better comparison was presented. The results demonstrate the advantage of parallelization, specifically using GPUs to speed up calculation of k-means up to 30 times and EM up to 17.9 times of a single threaded CPU. In the future, the effects of using shared memory on the performance will be studied. Additionally, running the algorithms concurrently on more than one GPU will be presented to show how GPUs can benefit from cross device memory access [10]. Additionally, the advantage of concurrent kernels and multi streams will be experimented. Furthermore, a comparison between the performance hits of single-precision and double-precision floating point calculations will be observed.

REFERENCES